# QUERY PROCESSING FROM MILITARY MAPS AND SAND MODELS (NON-GRAPHICAL QUERIES)

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

*By*

CAPT. K. SURENDRA NATH

*to the*

**COMPUTER SCIENCE PROGRAMME**

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

SEPTEMBER, 1983

# CERTIFICATE

Certified that the thesis entitled 'QUERY PROCESSING FROM MILITARY MAPS AND SAND MODELS (NON-GRAPHICAL QUERIES)' by Capt K. Surendra Nath has been carried out under our supervision and this has not been submitted elsewhere for a degree.

Aug, 1983

( S.G. Dhande )
Asst. Professor
Department of Mechanical Engineering and
Computer Science Program,
Indian Institute of Technology,
KANPUR

( R. Sankar )
Professor
Department of Computer Science,
Indian Institute of Technology,
KANPUR

CSP-1983-M-NAT-QUE

ACKNOWLEDGEMENT

It is with a deep sense of gratitude, I acknowledge the guidance provided by Dr. R. Sankar and Dr. S.G. Dhande. I shall remain ever indebted to them for their kind co-operation, patient outlook and encouragement throughout the development of the present work.

It takes a lot of ink to go from source to sink. It is with fond appreciation and pleasure that I recall my association with Capt A.V. Subramanian and Mr. Girish C. Elchuri. But for their efforts and help throughout the duration of my course, I would have been unaware of both the pleasures as well as the philosophy of programming.

It is with pleasure that I recall my association with Sundar J. Vaska, Capt D.S. Virk, Lt N.K. Rao, Kompella V. Rao, Sarvanan and all my other classmates. I would also like to acknowledge the help and guidance provided by Deepak and Gupta.

A special note of thanks goes to Capt Deepak G. Wakankar for all the help extended in patiently reading and correcting the manuscript. I am also thankful to Suresh Donepudi for his help in preparing the figures.

I would also like to thank all the faculty members of Computer Science Program for their help and co-operation

throughout the duration of my course.

I shall remain indebted to my wife and children. The spirit and understanding that was written all over them while greeting my return home after long hours of study, was truely splendid!

Last but not the least, I would like to acknowledge the excellent typing work of Mr. Mohammed Anwar.

K. SURENDRA NATH

## ABSTRACT

Development of a non-graphical, geographic query-processor system for use in computer-aided sand model simulations, is a necessity. In this work, a user friendly interactive query language (IQL) has been designed and developed. The IQL commands have been kept to a bare minimum and at the same time, any related query could be handled using this basic set of commands. An appropriate data structure for storing voluminous map data has been identified and implemented. The data has been so stored that it is amenable to easy and efficient access as well as retrieval. Run-time support routines for disk I/O have been developed and implemented in DEC-MACRO. Three representative modules namely, insert, update and query have been implemented.

LIST OF CONTENTS

# CHAPTER 1

## INTRODUCTION

The key to success of any military operation is, sound planning based on all available information such as, intelligence information, geographic information and meteorological information to mention a few of the most salient ones. Of these the geographic information is the only one that can be conveniently reduced to a pictorial form either by an equivalent clay model, a slide or a chart. These tools are being used extensively by commanders at all levels to finalise their operational plans. The most popular of these being a clay model called, "SAND MODEL".

Before any military operation, be it an attack, a defence, an advance or a raid, the area of operations is known before hand. Therefore, for planning purposes following are the courses open:

(i) All the necessary markings such as, deployment of troops and equipment, communications and logistics plan, could be inscribed over a talc sheet placed over the map/maps of the given area.

(ii) A model of the area could be prepared with clay and sand and thereafter various plans could be discussed to arrive at the best possible solution.

(iii) A combination of both the above courses.

The above courses have the following drawbacks:

(i) Both talc sheet marking as well as preparation of sand models are time consuming and error prone due to human handling.

(ii) Both these are very temporary in nature and there is no way of keeping a record of the old one when a new one is required.

(iii) They are unwieldy.

(iv) In fast changing tactical situations vital time may be used up in just preparing a chart or a clay model, reducing the actual amount of available planning time.

(v) No inbuilt capability for answering queries of any kind like, distance between places, intervisibility information for communication planning, fordability of a river, crossing points etc., without actual manual computations.

(vi) Prone to erasure and accidental damage even at the preparatory stage let alone the storage

stage.

(vii) No capacity for enlargement, if and when required for appreciation of finer details necessiated under special circumstances.

(viii) Life of a map sheet is reduced by a large extent due to folding and refolding to different sizes for different applications.

To overcome the above drawbacks, computer-aided sand model displays have almost become a necessity. Introduction and implementation of computer-aided sand model displays would be a giant step towards modernising the machinery of any standalone defence force.

## 1.1 Computer-aided Sand Model Displays

Recent advances made in the field of Computer-aided Cartography, have opened new vistas for numerous practical applications, one such application being in sand model displays. An aesthetic picture acts as a stimulant to the mind. For planning purposes, a pictorial display of the area of operations helps the mind to perceive better without wasting time on abstract imagination. Any thought that flashes across the mind could be immediately reduced to a pictorial representation on the screen. Once the pros and cons of this idea have been discussed, this

picture could be retained, or if needed it could be wiped out immediately. In a computer-aided sand model display, this sort of flexibility is available for the asking. This versatility is a major advantage of such displays which could be exploited to the maximum in relieving planning personnel of having to remember and imagine minor and petty details. Some of the other advantages that accrue from computer-aided sand model displays are:

(i) All the likely areas of operations could be digitised before hand and stored on bulk storage media. This should be adequately documented for easy understanding and speedy retrieval of information. Once this has been done, actual display of any required area to any required scale becomes an easy task. Since the digitised information is edited before storage, human error is eliminated.

(ii) Since bulk storage is available, copies of the old plans can be stored for future reference as long as they are necessary.

(iii) The display is projected onto a screen as large as required and takes very little time for projection, correction or editing and deletion.

(iv) Vital time otherwise lost in preparing a marked

chart or sketch or a sand model is now saved.

(v) A non-graphical database, supported by a user friendly command language, could be developed for query handling. Time wasted in manual computation is saved and the element of human error is eliminated.

(vi) Storage for long periods is safe and accidental damage is avoided.

(vii) Due to the concept of layering of information into different levels earmarked for different entities, it is possible to view only as much detail as required at any one time on a need to know basis. Scaling up or down and scrolling is possible.

(viii) Life of the actual base map sheets is enhanced. Many more advantages not listed above are inherent to these displays.

## 1.2 Scope of Present Work

The present work is in continuation of the thesis, "COMPUTER AIDED SAND MODEL PICTURE CREATION USING INTERACTIVE MODE" [1] and the thesis, "CARTOGRAPHIC CONSIDERATIONS IN COMPUTER-AIDED SAND MODEL SIMULATIONS" [2]. Various definitions and details relating to traditional

cartography, computer-aided cartography and the present state of art have already been dealt with at length in the previous work mentioned above. However, in order to get a complete picture, it is indeed essential to glance through the previous work.

The endeavour here is to implement successfully some of the suggestions and more, that have been highlighted in the previous work. These suggestions would form the broad spectrum on which the present work will be based and continued.

### 1.2.1 Objectives

(i) To develop a nongraphical query processor, supported by a user friendly command language for answering various queries which will be dealt with in the later chapters.

(ii) Use of Strip Trees, which are a hierarchical representation for curves for query processing [3]. This part is also being developed presently [7].

### 1.2.2 Organisation and Layout

The present work could be broadly divided into two main sections, namely:

(i) The non-graphical datastore section that deals

primarily with the organisation of physical data for extensive query handling.

(ii) Development of software to provide adequate runtime support for both loading and retrieving information from the datastore. The runtime support routines have been developed in MACRO (DEC-10 ASSEMBLY LANGUAGE) and the application program modules of query, insert and update have been developed in Pascal.

The details regarding organisation and design of the interactive query language have been dealt with in Chapters 2 and 3.

The physical data organisation is dealt with in Chapter 4. Program specification is explained in Chapter 5. Chapter 6 highlights some of the important algorithms used and Chapter 7 summarizes the present work and enumerates scope for future work.

# CHAPTER 2

## A NON-GRAPHICAL DATASTORE SYSTEM

Having listed out the objectives for the present work, which shall deal primarily with the design and development of a query-based geographical non-graphic query processor, we shall discuss the various levels of abstraction present in a typical database management system and look at its principal functions. We shall also discuss a "real world" model having a capability to store and manipulate real data. This model called the "Entity-relationship" model will be discussed in the succeeding chapters. The present work is based mainly on the above model. However, the present work cannot strictly be classified as a database system. Some of the principles of a database system have been used to advantage in the present work.

### 2.1 A Database System

Geographical map data is voluminous. We call such voluminous data that is stored more-or-less permanently in a computer, a "DATABASE". The software that allows one or more persons to use and/or modify this data is a

"database management system" (DBMS). One of its major roles is to allow the user to deal with the data in abstract terms, rather than as the computer stores it. In this sense, the DBMS acts as an interpreter for a high level language. There are many other functions that can and should be carried out by the DBMS; to name a few [4]:

(i) Security: Access to data should be restricted and strict control exercised.

(ii) Synchronization: The DBMS should provide protection against inconsistencies, that result from two approximately simultaneous operations on a data item.

(iii) Crash protection and recovering: There should be facilities to make regular backup copies of the database and to reconstruct the database after a hardware or software crash.

## 2.2 Data Independence

Applications which are interactive present a number of requirements on database management systems [5]. From an application programmers point of view, the database system should provide support for a simple tabular representation of data, to include its syntax and

semantics. This is very essential to provide independence between the data and the application programs. Data independence makes it possible to write programs which can use different data tables.

The basic requirement of database systems is that the application programmer should be able to write application programs that are independent of the data stored in the database.

Data used by an application program has four components:

(i) The data itself

(ii) The syntax (format) of the data

(iii) The semantics (meaning) of the data

(iv) The access path to the data.

Usually the second, third and fourth components are implicitly incorporated into the program and the database only keeps track of the data itself. The format of the data, its meaning and its access path must be known to the application program.

If all these components of the data are stored in the database and the data can be retrieved by name (knowledge of access path not needed), the data is said to be "self-describing". If data is self-describing, then the application program need not keep track of the data's

syntax, semantics, or access paths. Self-describing data is a necessary ingredient in achieving independence between programs and data.

## 2.3 Database Organisation

Two basic questions that arise at this stage are:

(i) What are the appropriate data structures with which to implement a typical physical database?

(ii) What are the properties or attributes of useful data and how should they be represented by physical structures?

In order to answer these questions, we introduce an informal model called the entity-relationship model of data. This model will serve to justify the kinds of data structures which will be used. More details on this aspect will be dealt with in Chapter 4.

Before we proceed further we shall define two terms which will be used extensively during the course of this work, namely:

(i) Entity: Entity is a thing that exists and is distinguishable; that is, we can tell one entity from another. For example, river is an entity and so is a road. A list of all the

entities encountered in our application along with their characteristics (attributes) are listed at Appx. I.

(ii) Attributes: Entities have properties called attributes, which associate a value from a domain of values for that attribute. Usually, the domain of an attribute will be a set of integers, real numbers, character strings or a combination of the above, but we do not rule out other types of values.

Having defined entities and attributes, we shall now organise the database to consist of a set of tables which are nothing but a collection of relationships between entities. Each table is an ordered set of records. Each record consists of a collection of named fields. Each record is a row in the table and each field is a column in that table. Information regarding the length of each field (denoting an attribute) is also stored in the database in the form of another table along with the data.

Having realized this model thus far, the most important thing at this juncture is to design and develop an interactive query language (IQL) for handling all types of queries that are likely to be encountered.

Having realized this, the final stage would be to develop and implement the application programs that are necessary to answer any type of query that may be raised.

# CHAPTER 3

## INTERACTIVE QUERY LANGUAGE (IQL)

This chapter highlights the design and syntax details of the interactive query language being used in support of the present work. The first step towards designing any language would involve the enumeration of its grammar or its syntax. Keeping in view the requirements of a sand-model discussion or exercise and the types of queries encountered, an IQL has been developed. After analysing all the likely queries, these have been grouped or combined to arrive at an optimum list. With this optimum list, any type of query can be handled with the help of the commands that have been developed.

In this chapter all the commands that will be used for handling any type of query have been listed out. For each query command, a brief description has been given which emphasises both the purpose as well as action taken by the system. All the query commands have been explained in the order they are likely to be given, wherever possible. Wherever more than one combination of a basic command is likely, not all such combinations have been listed. Also, all the commands are enumerated

in alphabetical order, in the form of a table at Appendix II. The appropriate error messages are listed at Appendix III.

## 3.1 Grammar

The grammar of any language is a 4-tuple. In this work, the syntax of the IQL is specified using the Backus-Naur Form (BNF). We specify the grammar of IQL written as G(IQL), by specifying the individual members of the 4-tuple, viz, T, N, P and S.

⟨ENTITY⟩ ::= ROAD|RIVER|CANAL|RLYLINE|
RLYSTNS|BRIDGE|VILLAGE|
WATERRES|AIRFD

⟨entity name⟩ ::= ⟨identifier⟩|⟨integer⟩

⟨identifier⟩ ::= ⟨letter⟩ ⟨letter or digit⟩

⟨letter or digit⟩ ::= ⟨letter⟩|⟨digit⟩|⟨letter⟩ ⟨letter or digit⟩|⟨digit⟩ ⟨letter or digit⟩

⟨attribute⟩ ::= ⟨identifier⟩

⟨OP⟩ ::= <|>|<=|>=|=|#|

⟨Value⟩ ::= ⟨Integer Const⟩|⟨Real Const⟩|
⟨Boolean Const⟩|⟨String⟩

⟨String⟩ ::= ⟨letter⟩ ⟨string⟩|⟨letter⟩

⟨Boolean Const⟩ ::= True|False

⟨location⟩ ::= ⟨X-coord⟩ ⟨Y-coord⟩

⟨X-coord⟩ ::= ⟨Real Const⟩

⟨Y-coord⟩ ::= ⟨Real Const⟩

⟨Relop⟩ ::= And/or

⟨Subexp⟩ ::= ⟨attribute⟩ ⟨op⟩⟨value⟩

⟨Exp⟩ ::= ⟨Subexp⟩|⟨Subexp⟩ ⟨relop⟩
⟨Exp⟩|(⟨Exp⟩)

## 3.2 IQL Commands

For any type of query to be processed, certain items are common to each and every query. These items will be referred to as the default set and are as follows:

(i) The Map Number,

(ii) The range or area of search,

(iii) The name of the entity under question.

These three items are set to their respective default values at the very outset. The relevant commands for setting these three defaults are:

MAP:

RANGE:

ENTITY:

Against Map, the map number associated with the area under query is typed in. Against Range, the range

in which a query is being sought is typed in. Here we impose one restriction on the user, that is, to specify range only in terms of grid squares. If the rnage is only one grid square, a four figure grid reference as XMIN XMAX YMIN YMAX, will suffice. If it is more than one grid square, then, the lower left and upper right points are given. For the entity, the name of the entity is typed in after ':'. These defaults can be changed at any stage. As long as this change is not affected, the most recently set default holds.

The next set of commands are:

HISTORY

HISTORY/⟨attribute code⟩

LIST

LISTALL

NUMBER

NUMBERALL

LIST ⟨attribute⟩ ⟨op⟩ ⟨value⟩ (AND / OR) ...

NUMBER ⟨attribute⟩ ⟨op⟩ ⟨value⟩ (AND / OR) ...

LIST/ ⟨entity⟩ ⟨attribute name⟩ ⟨op⟩ ⟨value⟩
OR
NUMBER AND, ...

LISTALL/ ⟨entity⟩

NUMBERALL/ ⟨entity⟩

On setting the defaults, "HISTORY" lists out all the attributes of the entity under question in two parts. The first part contains all those attributes which do not change with range. For instance, if the entity was a road, attributes such as the type of highway it is, the number of lanes, metalled or non-metalled etc. do not change (generally speaking) with range, whereas attributes such as, bridges and level crossing etc. are different in different ranges. All such attributes are listed under variants. If this command is followed by a slash and an entity or attribute code, then the attributes of the relevant item are listed out. In certain situations, the answer required for a query is one or more entities itself. In this case while setting the defaults against Entity, the ':' is followed by a '?'. For instance, if one would like to know all the towns in a given range satisfying certain conditions, the command could be, LISTALL population = 1000 AND RS = True, Here, the system would print out all such towns along with their locations that satisfy the above condition. Any command prefixed with "LIST" can safely be used whenever the default entity has a '?' following a ':'. In this context, if "HISTORY" were to be used instead of "LIST", it would result in an error. When the command "LIST" is followed by a slash and then the entity followed

by a certain query; here again there are two different methods of processing. If the entity default was already set and not a '?', this command lists out all those entities which satisfy the given conditions. For instance, if entity default was a road say NH1 and the command was 'LIST/BRIDGE Width = 10 AND clearance = 20,' here, all such bridges on this road are listed. If the entity was ':?'then, this would result in listing all the roads in that range whose bridges satisfy this condition. Also, after echoing this answer the system goes into a special mode and awaits input on the prompt '@'. Here we can ask for the history of any of the entities listed. In case no further clarifications on any of these entities are required, we could leave this special mode and get back into the query mode by just typing a 'Ø'. This facility has been developed to help the user for asking the history of any one of these entities. If this provision were not built in, then, on getting a list of all entities that satisfied some given condition, the user would have to go back and set the entity default and then ask for history. With this additional facility of a special mode, this inconvenience is overcome. The special mode facility is available only in this case and in no other case or instance.

In all the above list of commands if the prefix "LIST" is substituted by "NUMBER", action taken is the same except for the fact that, in this command only the number of entities are echoed and not their names and locations. For e.g., if in a given range, there are bridges 1, 2 and 5 the command "LIST" would have typed the name of each bridge and its location (Six Figure grid reference) whereas the command "NUMBER" would only have displayed the integer '3' on the screen. The treatment meted out to the command "NUMBER", is exactly the same as that of "LIST" except for the difference explained above. Command "NUMBER" is valid wherever "LIST" is.

The commands that have been listed thus far are more than adequate to handle any type of query that may be raised. Apart from these, there are certain other commands which are of a general nature, but nevertheless are most necessary. These commands are as follows:

AGAIN

ENTITIES

ATTRIBUTES/ <entity name>

HELP

EXIT

The command "AGAIN", displays the answer to the current query. It may so happen that the list of

answers may be too long. Hence, it may be difficult for a person to comprehend all this information in one scan. In all such cases the command "AGAIN", would be most useful.

The command "ENTITIES", lists out all the entities that are present in the system.. The command "ATTRIBUTES" followed by a slash and then the entity name, results in typing out all the attributes that form a part of that Entity. For instance, "ENTITIES" prints out the complete list as River, Road, Canal etc., and for ATTRIBUTES/Road, prints out all the attributes that are associated with a road.

"HELP" is a command that results in printing out the complete help file that has been created and merged with this package. The help file contains a list of all the basic as well as all those combinations of commands derived from the basic commands. This file also contains a large number of examples with each command and is quite exhaustive and self-explanatory.

Some of the important algorithms used in this chapter will be dealt with in detail in Chapter 6. Hence, no mention whatsoever has been made regarding the algorithms. Finally, one other facility that has been provided is, the answer to every query is displayed

along with the values of the three defaults that were set. It is felt that with the introduction of this facility, the user has a ready reckoner always in front of him, which enables him to carry out his dialogue with the program without having to remember by heart the default set.

# CHAPTER 4

## DATA ORGANISATION

One of the prerequisites of any query processor system is, to store and retrieve voluminous data that is present in the fastest possible time and in an efficient manner. This chapter deals primarily with the organisation of data at the physical level, namely, the secondary storage.

A file consists of a series of records. The length of a record varies, depending on the type of information it stores. Map data pertaining to a single map is extracted and loaded into the data store. Data pertaining to a single map goes into one unique file. This file is then closed and a new file opened for storing data pertaining to another map. Thus, a file is a sequence of varint records and is unique for any single map.

For disc I/O handling, this package takes recourse to an I/O routine developed in Macro [6]. The above mentioned routine has been suitably modified to cater to the requirements of the present work.

## 4.1 Page Organisation

Each file corresponding to a single map is divided into a number of logical blocks called, "PAGES." These pages contain variable length records and are uniquely identified by their page numbers, starting from 0. The size of the data store is generally referred to in terms of the number of pages it houses. The basic unit of retrieval from secondary storage (disc) is in terms of blocks. (Each block in Dec-10 has 128 words).

Three types of pages will be dealt with in this section. System page and Delete page are of length 1 block each (128 words) and the rest of the pages are of length 2 blocks (256 words) each.

### 4.1.1 System Page

System page organization is shown in Figure 4.1 Bitwise breakup of the DBkey is shown in Figure 4.2. The right half word of the system page (bits 18 to 35) contains a dbkey which points to the first occurrence of that entity in the datastore. Bits 13 and 14 called "SYNMTY", indicate that the name is a string of characters if it is a '0' and that it is a number if it contains a '1'. Record length specifies the average length of the record less the words reserved for headers and name. Locations 1 to 9 indicate the type of entity.

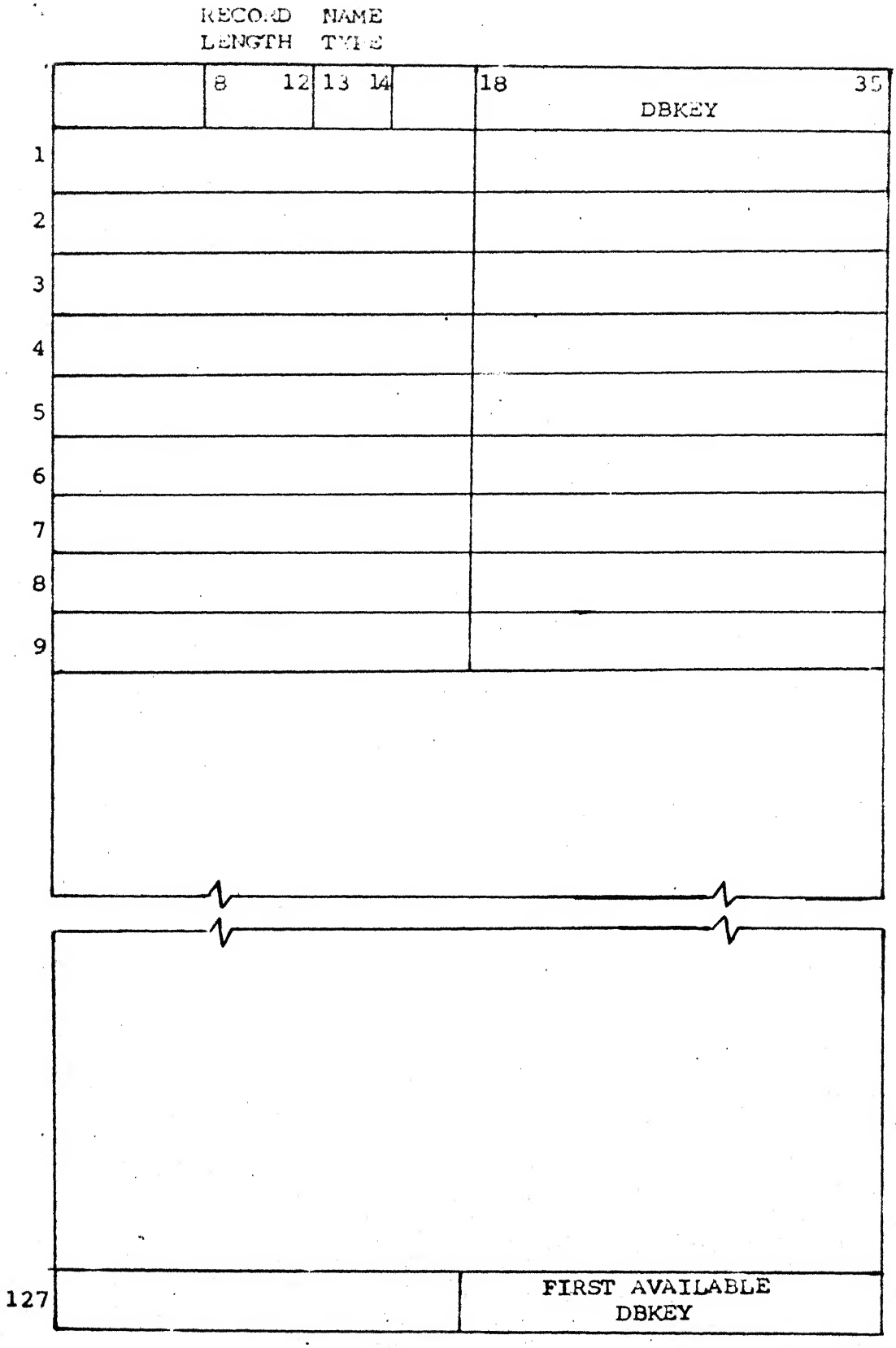



FIG 4.1 SYSTEM PAGE

(Road, River etc.) locations 100 to 105 store in the left half the length of each record in number of words in that type of array. The right half stores the address of that record (determined by the index) or array declared in Pascal. This address is referred whenever information is being transferred into Pascal record or array. Word 127 of system page stores the first available dbkey in the format shown in Figure 4.2.

### 4.1.2 Delete Page

Delete page organisation is shown in Figure 4.3. Bit 0 is the valid bit. A 0 in this field indicates no entry and a 1 indicates a valid entry. Bits 4 to 17 are for page number and 18 to 35 are for words left in that page. Delete page is made use of whenever a new dbkey is asked for. Macro routine "getdbkey" makes use of this in allocating a new dbkey on request. Algorithm for this allocation is given in detail in Chapter 6.

### 4.1.3 Data Page

Apart from the system page and delete page which are of 128 words each, (1 block) the rest are datapages. This consists of a page header at the top followed by 14 line headers and the rest of the lines are used for actual data. Page configuration is shown in Figure 4.4.


FIG 4.2 DBKEY

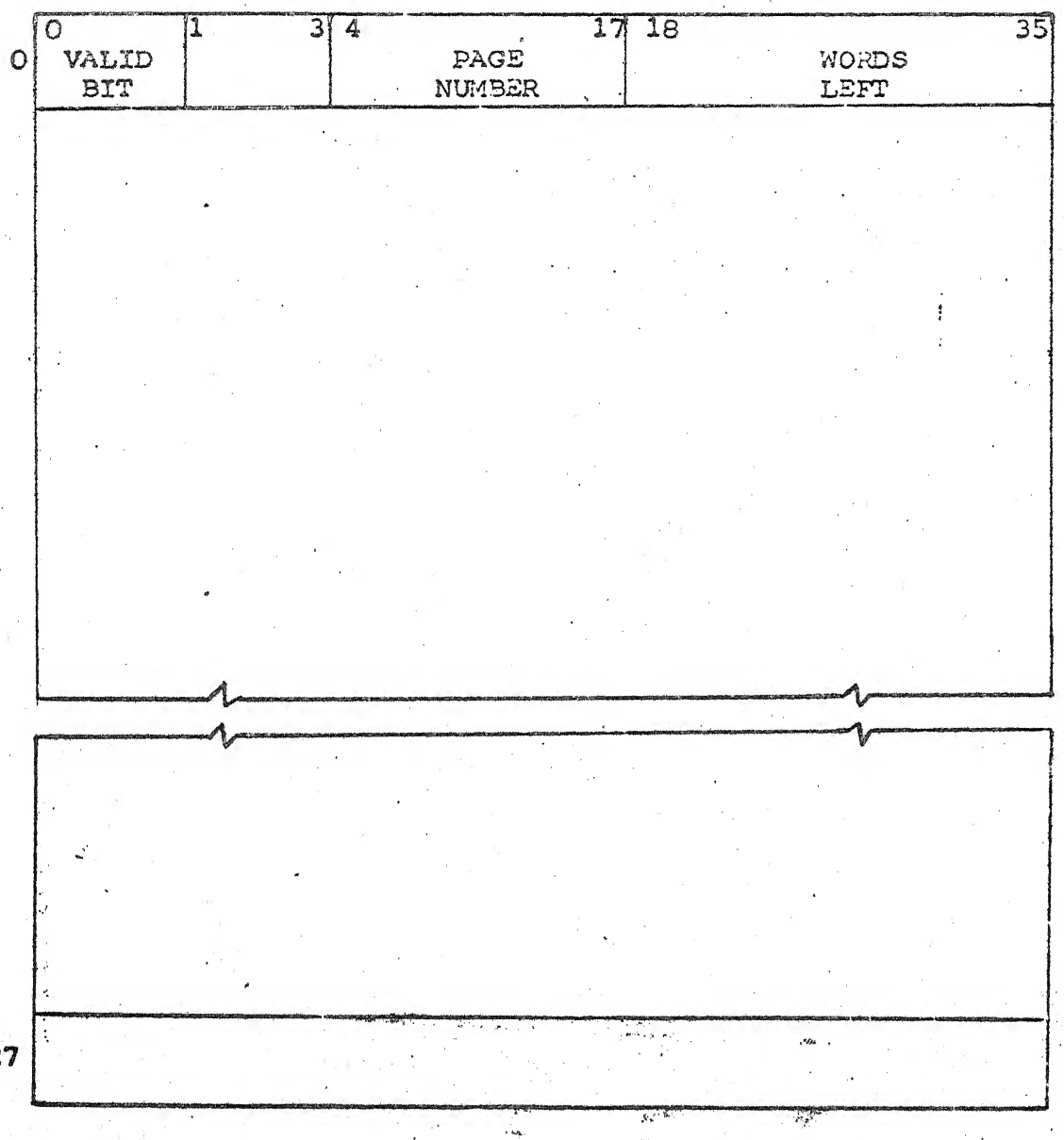



FIG 4.3 DELETE PAGE

Format of headers (Page, line and record) is shown in Figure 4.5. Average length of record has been calculated from the data generated for this work as a sample from an actual map. The number of lines per page have been calculated based on this average record length. These calculations are approximate and are based on sample test data. These calculations are given at Appendix IV attached.

Following are the fields in Page header:

(a) Page No. - contains current page number.

(b) First free word - address of 1st free word.

(c) Words left counter - number of words left in that page.

(d) Lines left counter - number of lines left in that page.

(e) Page full bit - status of page.

Following are the fields in line header:

(a) Start address - gives the starting address of the record as displacement from the top of that page.

(b) Record size - gives the length of that record.

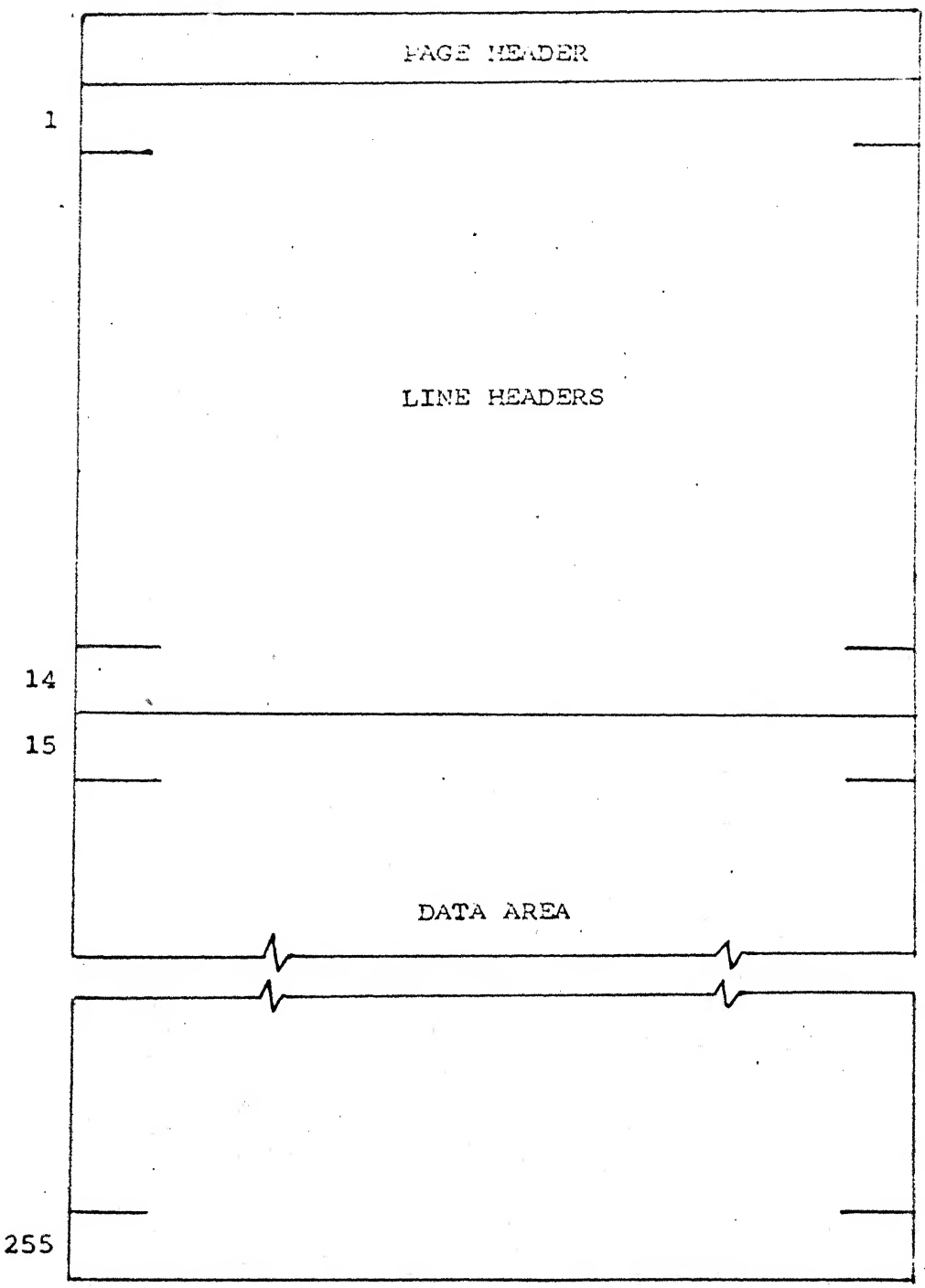



FIG 4.4 PAGE CONFIGURATION

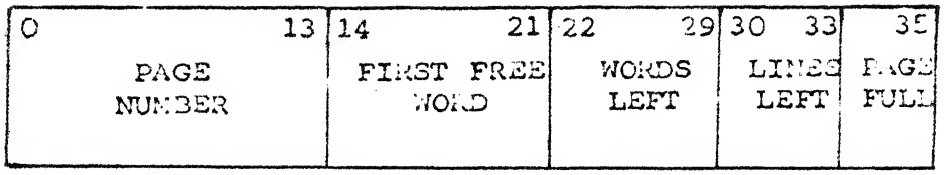
Following are the fields in record header:

Entity number - contains the Index corresponding to the entity that is being referred.

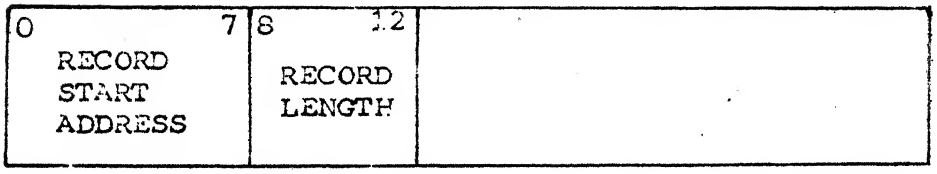
Record size - contains the record size in number of words.

## 4.2 Lengths of Fields

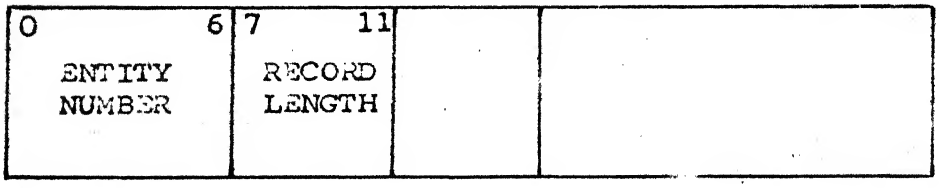
As is evident from calculations attached at Appendix V, the number of lines per page is 14. Hence, 4 bits are alloted for the line number and 14 bits for page numbers. The size of the datastore generally expressed in terms of pages, can therefore grow to be as large as $2^{14}$ (16K pages). This is sufficiently large and is considered more than adequate for storing the information pertaining to a single map. Each page consists of 2 blocks each (256 words). Hence 8 bits have been alloted for both the first free word and words left counter fields. Maximum record length pertains to the Entity "VILLAGE", occupying 26 words and the minimum length pertains to the entity "ROAD", that occupies roughly 9 words. Hence 6 bits have been allocated for record length. These fields have been clearly shown along with their sizes, in Figure 4.5.




PAGE HEADER




LINE HEADER




RECORD HEADER

FIG 4.5 FORMAT OF HEADERS

The macro routines providing the run-time support for this package have been listed as external in the main listing. Some of the important algorithms have been discussed in Chapter 6. These routines, in fact, act as an interface between the physical data and the application program.

# CHAPTER 5

# SPECIFICATION OF PROGRAM MODULES

The application program basically consists of four main modules whose specification and implementation details are given in the following paragraphs.

## 5.1 Organisation

The package has been divided into the following modules:

(i) Overall tree structure of the package (Figure (5.1))

(ii) Query Module (Figure (5.2))

(iii) Inser-update module (Figure (5.3))

(iv) Delete module

In this chapter, some of the important aspects of each module will be explained. However, the actual algorithms will be explained in detail in Chapter 6.

## 5.2 Query Module

The actions performed in this module are as follows:

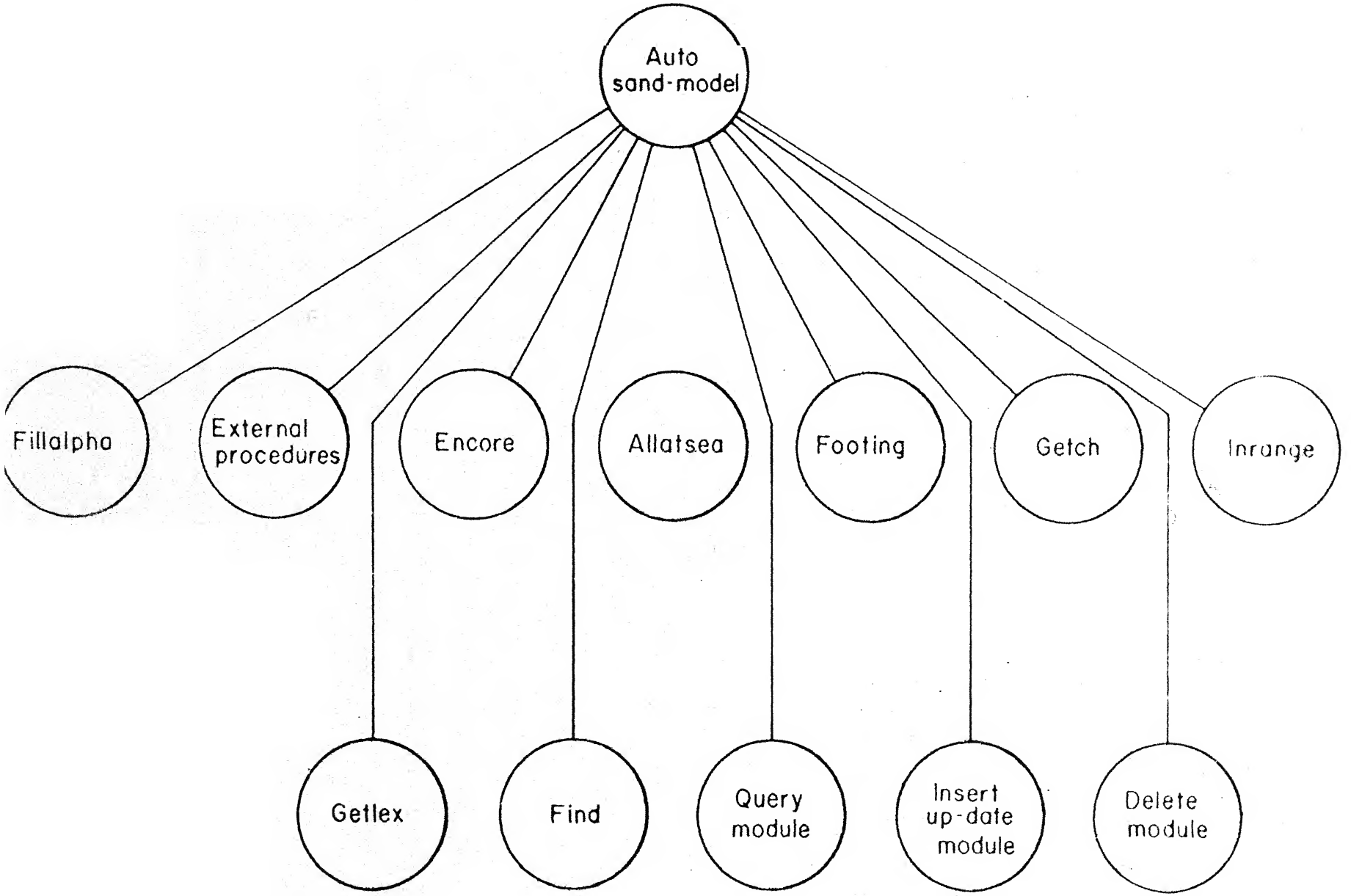



Fig.5.1 Overall tree structure of the package

(i) Lexical analysis and parsing

(ii) Default setting

(iii) Range checking

(iv) Flashing "ENTITIES" and "ATTRIBUTES"

(v) Processing commands "HISTORY" and "HISTWITH-CODENO."

(vi) Processing commands "LIST" and "NUMBER"

(vii) Processing commands "HELP" and "AGAIN"

(viii) Error recovery and Error message handling.

A brief explanation of each of the above eight actions performed in the query module are given in the succeeding sub-sections.

All the entities present in the datastore, their attributes and the type information of each attribute have been initialised through init procedures. Also, all the commands mentioned in Chapter 3 earlier (IQL commands) and the defaults have also been initialised through init procedures. These initialisations prove very handy while searching is done to locate an attribute, determine its attribute type and for comparison purposes.

### 5.2.1 Lexical Analysis and Parsing

Procedure "GETLEX" reads each command from the terminal interactively, fills the same into a line array

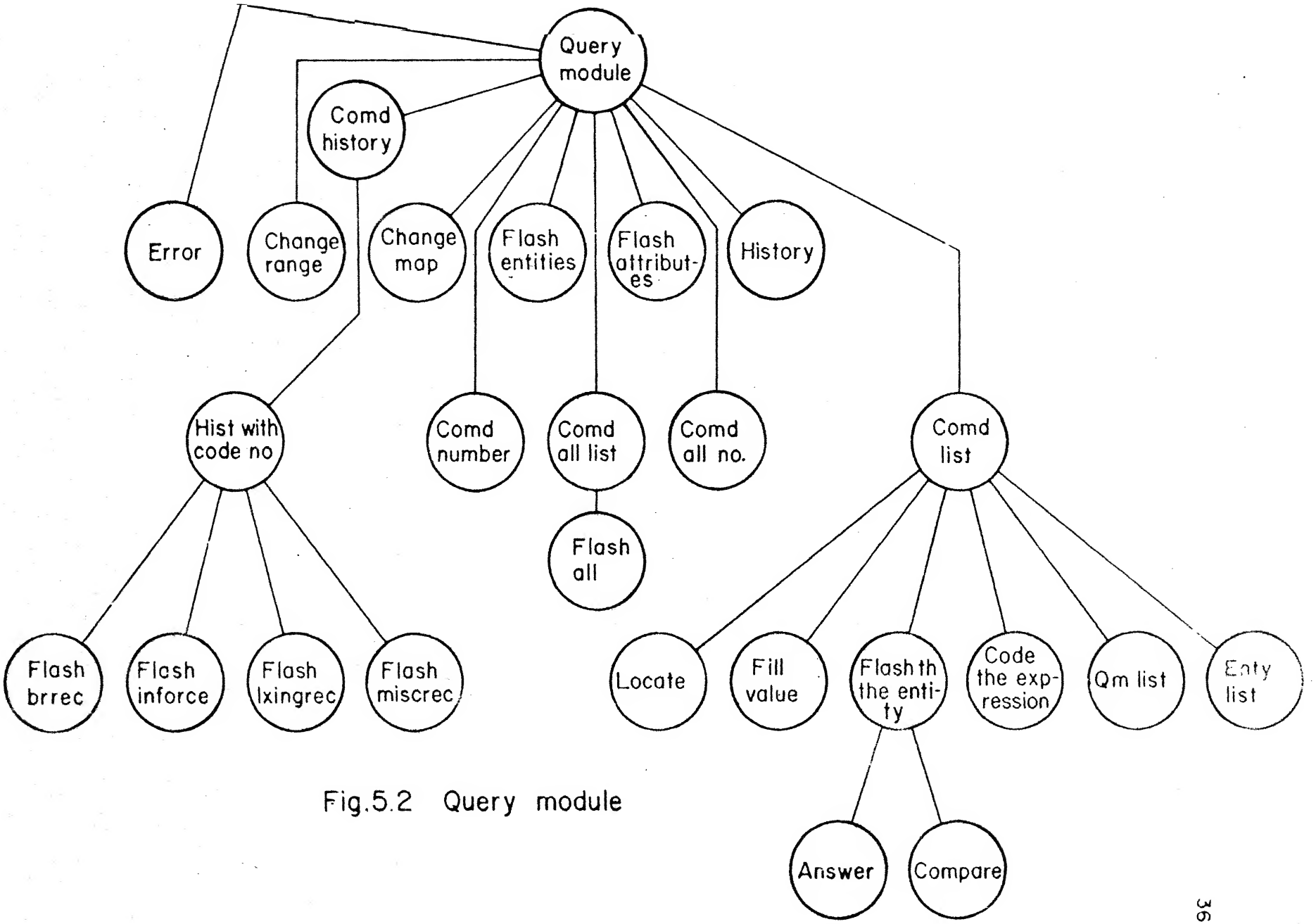



Fig.5.2 Query module

and appends a '*' at the end of each line indicating the end of any command. Procedure "GETCH", then reads character by character until it meets a delimiter. The set of characters read are then assigned to a variable called "LEX".

Each of these leximes or tokens are then compared with standard commands that have been stored and initialised in the corresponding init procedures. On successful comparison, these commands are suitably interpreted and control is transferred to an appropriate procedure or function, which outputs the result onto the screen. In case a certain comparison fails, an appropriate error message is flashed onto the screen and the program awaits further input on the query mode prompt, a '?'. The program remains in this recursive loop until it meets an "EXIT".

### 5.2.2 Setting Defaults

The pre-requisites before processing any command are, Mapnumber, Range and Entity (name or number). These defaults are set using procedures "CHANGEMAP" and "CHANGERANGE" and the entity index is saved in "SYSTEMCURRENT". Once these defaults are set, they are assumed to be common for all the queries that follow, unless and until the user changes these during the course of program execution.

### 5.2.3 Range Checking

Range checking is done at two different stages. Initially, when the range is specified, it is checked against the range field pertaining to the default map under query. Subsequently, whenever a query is being processed during program execution. The answer to a query is flashed only when function "INRANGE" returns a true value.

### 5.2.4 Flashing Entities and Attributes

To give an overview regarding the entities present and their respective attributes, procedures "FLASHENTITIES" and "FLASHATTRIBUTES" are used. These two procedures primarily use the arrays that have been initialised at the beginning of the program run through initprocedures.

### 5.2.5 Processing "HISTORY" and "HISTWITHCODENO."

Depending on the defaults that have been set, procedure "HISTORY" when called, is supplied with the entity index stored in "SYSTEMCURRENT". Depending on this index the appropriate case variant is processed. Before this, we call on a macro routine "GETOCCURRENCE" which returns entity occurrence dbkey. Using this dbkey we fill the pascal record buffer with the information pertaining to that entity. Thereafter, the rest of the details

that are flashed onto the screen are taken care of in the relevant case variant portion of procedure "HISTORY".

Procedure "HISTORY" lists out information pertaining to only the non-variant attributes. The variant attributes at this juncture are only flashed onto the screen and no values are mentioned against them. To get a better insight into their values, a slightly different methodology has to be followed which will be explained in a later section.

Procedure "HISTWITHCODENO" is evoked when the history of a certain variant attribute listed while processing the command "HISTORY" is required. To processes this command, the following information is require to be updated:

(i) a global array called "VARARRAY", which is an array [1...10] of integers, stores the dbkey to the first occurrence of a variant attribute in the datastore depending on a unique index supplied. For instance in the case of a road, VARARRAY [1] stores the dbkey to the first bridge occurrence on the road, VARARRAY [2] stores the dbkey to the first village occurrence on the road and so on.

(ii) the index to any variant attribute listed in procedure "HISTORY" is passed as a parameter

to procedure "HISTWITHCODENO", which decides the appropriate case variant to which the control will go.

(iii) The index number to the pascal record array is also determined. This number is also a unique one and depends on the type of record the information is stored in, viz. Bridge-record, levelcrossing record, inforecord or or miscellaneous record.

Thereafter, following are the steps taken to answer the query:

(i) A call is made to a macro routine Fillsuccessive records, viz. "FILSUCRECS". Vararray index and the index number indicating the type of record are passed as parameters.

(ii) Once the pascalrecord array is filled, a call to one of the procedures "FLASHBRRECINFO", "FLASHLXINGRECINFO", "FLASHINFORECINFO" or "FLASHMISCRECINFO" does the required type and range checking and flashes the required answers onto the terminal.

(iii) This process is stopped when in the last word of the pasrecbuffer a '-1' is encountered. "FILSUCRECS" ensures that the last variant in

that chain has its nextentity pointer field (last word in that record) initialised to '-1'.

Adequate documentation has been done in program listing and appropriate comments have been included wherever necessary for ease of understanding.

5.2.6 Processing "LIST" and "NUMBER"

These two basic commands are only a sequel to many possible combinations that can be constructed from them. Processing these two commands is by far the most complex part of the query module. Some of the possible combinations are as follows:

(i) LIST <EXPRESSION> <Relop> <EXPRESSION>

(ii) LIST/< entity > <EXPRESSION> < RELOP> < EXPRESSION >

(iii) LISTALL/< entity>

In any of the above three combinations "NUMBER" could be substituted for "LIST" and "NUMBERALL" for "LISTALL". In combination (i), against entity default a '?' should be present, failing which an error message is flashed. In combination (ii), a '?' against entity default results in an error. For processing an expression Procedure "CODETHEEXPRESSION", is called. The given expression is coded by reading the expression

from left to right. While coding, the Expression array (EXPARY) is appropriately updated. After updating, if against entity a '?' was present, a call to procedure "QMLIST" flashes back the answers. Otherwise, a call to procedure "ENTYLIST" flashes back the answers. From both "QMLIST" and "ENTYLIST", a call to procedure "FLASHTHEENTITY" does the actual process of locating an attribute and filling that attribute value in codedexp record by a call to procedures "LOCATE" and "FILLVALUE" which are inside procedure "FLASHTHEENTITY". A call to functions "COMPARE" and "ANSWER" which are functions inside procedure "FLASHTHEENTITY", returns the final answer which is flashed onto the screen.

The command "NUMBER" when substituted for "LIST" does the same job as list, except that instead of flashing all the answers in an enumerated form it returns an integer which is the total number of entities that satisfied the given condition. For instance, if "LIST returns the names and locations of Bridges 1, 2, & 3, the command "NUMBER" would have returned the integer "3" as answer.

The command "LISTALL/ ENTITY" results in an error if against Entity default, a '?' was present. The processing done here is much simpler than in "LIST". Here,

no expressions are allowed after "/<entity>". After determining the systemcurrent (Entity index) the relevant record starting from the dbkey (extracted from system page) is filled into the pasrec array by a call to "FILPASRECBUF". Thereafter, a call to procedure "FLASH-ALL" flashes back the answer onto the screen. As in "LISTALL", "NUMBERALL" only returns an integer value.

In all the above discussed combinations, except "NUMBER" and "NUMBERALL", the system goes into a special mode called '@' mode. From here, the command "HISTORY/entitycode" is valid, which transfers control to procedure "HISTWITHCODENO". This facility has been provided for reasons discussed in Section 3.2. In case the user has no requirement of asking for "HISTORY", he could leave this mode and re-enter the query mode by typing a '0' on the '@' prompt.

5.2.7 Processing "HELP" and "AGAIN"

A help file has been created and is stored in the system. When called, the complete contents of this file are flashed onto the screen. In this file all the IQL commands for use in query and the commands for use in insert-update have been listed. Numerous examples of both, expected input by the program as well as the expected output by the user, have been unambiguously mentioned.

### 5.2.8 Error Message Handling

Procedure "ERROR" is a local procedure and is duplicated in both the query module as well as the insert-update module. These error messages have been kept as crisp and concise as possible. When called, control is transferred to the appropriate case variant which flashes the relevant error message. Thereafter, control goes back to the next statement from where this procedure was called.

## 5.3 Inser-Update Module

The three primary operations to be performed by any query management system are:

(i) Insert

(ii) Update

(iii) Delete

The first two operations of both insert and update have been incorporated by using one procedure only. One boolean variable which has been declared globally is made use of to differenciate between these two operations. For geographic map applications, data once created and loaded into the datastore, remains resident almost permanently. Deleting an entity occurrence however, is a very rare phenomenon, though it cannot be totally

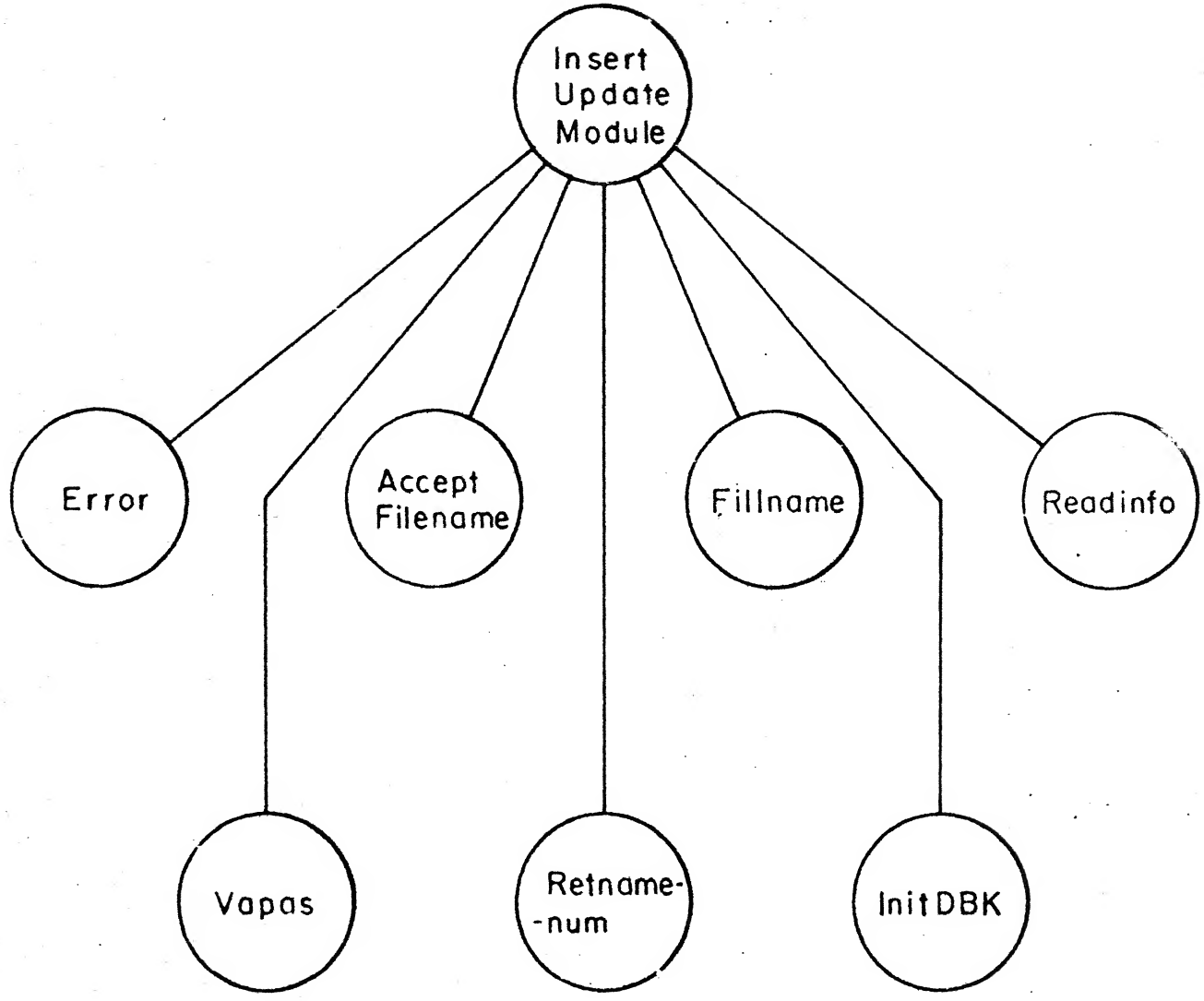



Fig.5.3 Insert-Update Module

ruled out. A delete facility, logical or otherwise, has therefore not been built into this package. However, space for the same has been created and left vacant in the program listing.

Both the Insert and Update operations will be carried out through an input file. The datastore will be loaded by reading and appropriately interpreting the information contained in this file. The update operation is also carried out by appropriately interpreting the information provided through a file.

The actions performed in this module are as follows:

(i) directory updating

(ii) loading the data

(iii) updating the data

(iv) error recovery and error message handling.

5.3.1 Directory Updating

The mapnumber, its range (XMIN, XMAX, YMIN, YMAX) and the filename in which the map information is made available, are ascertained in procedure "ACCEPTFILENAME". The directory stored in system is initialised accordingly. This information would be a vital factor, when a user is in the query mode, for carrying out range checking.

5.3.2 Loading the Data

As mentioned earlier, data could be loaded only through a file. No interactive loading facility has been provided. The insert operation is done in two phases. In the first phase, only the non-variant attributes of each entity are loaded. Procedure "FILLNAME" reads the entity type and entity name from the input file. A call to Macro routine "GETOCCURRENCE" returns the dbkey of the entity occurrence. If this dbkey is $\neq$ '0', an error message is flashed. Having ascertained that the entity being loaded into the datastore is indeed a fresh Insert operation, by a call to "GETOCCURRENCE", a call to "GETDBKEY" returns the value of the first available dbkey after searching through the delete and system page. The algorithm for getting a dbkey will be explained in Section 6.1. Initialisation of headers to the appropriate values is done in "GETDBKEY", before the actual dbkey is returned.

Information regarding attribute values is filled into the pascal record buffer from the input file and a call to "STOREREC", which is also a macro-routine, stores the information just read into the datastore, starting from the dbkey value returned earlier.

For loading the variant attribute values which forms the second phase of loading, the token "VAR ≠ No" is written in every line preceding any set of variant attributes. A call to "FILLNAME" identifies the name of the entity. This information is saved. The pasrec buffer is filled with variant attribute values. A call to procedure "INITDBK", updates the relevant pointers which were left vacant earlier while inserting the non-variant attribute values. This information is stored back in the datastore by a call to "PUTARRAYBACK" from function "READINFO".

The above process is repeated until the program encounters an "END" in the input file. Action is taken to close the data file by a call to "CLOSDB" on seeing "END". A typical input file containing "INSERT/UPDATE" information is at Appx V attached. This file contains Insert information pertaining to both the non-variant as well as the variant attributes.

### 5.3.3 Updating the Data

As mentioned earlier, both the insert and update operations are being carried out by using the same procedure, "INSERTUPDATE". The tasks carried out by both insert and update operations are similar in nature. The runtime support routines they evoke are almost the

same except for minor variations.

While updating information housed in the datastore, a call to "GETOCCURRENCE" should return a non-zero value unlike in insert where, "Getoccurrence" should return a zero. Having got the dbkey of the entity occurrence, a call to "SAVEREC" transfers the relevant record into a temporary buffer. Thereafter, the relevant record from Pasrecbuffer is transferred into temporary buffer overwriting the information already present in the temporary buffer. A call to "ENTRDB" stores back this record in the datastore.

One restriction has been imposed on the user at this juncture. Even if the user wants to change only a single attribute value, all the values have to be keyed in. This restriction has been imposed primarily for the purpose of optimising the amount of search and type cheching that would have otherwise been carried out. The number of attributes for each entity generally vary between 3 to 6 except in one case, where the number of attributes are 21. This restriction therefore, is not too stringent and was imposed purely from the point of view of striking a viable tradeoff between time complexity and ease of program handling.

Transition from insert to update and vice-versa is being achieved by setting a boolean flag "UPDATE"

to the current mode. Polling of this flag prior to every fresh entry into this program segment ensures the transfer of control to the right operation.

### 5.3.4 Error Message Handling

Procedure "ERROR" is a local procedure and error messages are handled in exactly the same way as explained before in Section 5.2.8.

# CHAPTER 6

# ALGORITHMS

This chapter highlights some of the important algorithms that have been used and the execution sequence.

(i) algorithm for getting a dbkey.

(ii) algorithm for filling pasrec buffer

(iii) algorithm for filling successive records

(iv) algorithm for storing a record

(v) algorithm for coding and evaluating an expression

(vi) Execution sequence.

Reference to the use of these algorithms has already been made in the earlier chapters.

## 6.1 Algorithm for Getting a dbkey

Step 1: Search through delete page entries while bit 0 is $\neq 0$. If words left against an entry are less than words required, repeat Step 1.

Step 2: Save page number. Substract words required from words left. If result is less than 9, go to

Step 5.

Step 3: Substract 1 from lines left. If result is less than 1, go to Step 5, else stop.

Step 4: Initialize headers

first free word := first free word + record size;

words left := word left - record size;

lines left := linesleft - 1;

If ((words left < 9) or (lines left < 1)) then

set page full bit := true

Update delete page if not done earlier. Stop.

Step 5: Remove that entry from delete page. Push up rest of the delete page entries. Set to '0' bit 0 of the last dummy entry. Go to Step 4.

Step 6: Since no page is found in delete page, pick out dbkey from word 127 of system page. If dbkey is 0,1 then go to Step 8 else, update the available dbkey field in word 127. Access that page. If words left is less than words required go to Step 7, else go to Step 4.

Step 7: Include this dbkey in the last entry in delete page. Increment page number by 1. Save this page number and line number '1' in the stack. Initialize word 127 of system page to line

number 2 and Page number to Page number +1. Go to Step 4.

Step 8: Word 127 is initialized to page 0 and line 2. Put page 0 and line 1 on top of stack. Go to Step 4.

## 6.2 Algorithm for Filling Pasrec Buffer

For the purpose of answering any query, information regarding the attributes of an entity are required to be present in a pascal record. Once this is ensured, each field of this record can be easily accessed by the pascal program. Starting address of the pascal record array for each entity is initialized before the program run, while loading the datastore.

Step 1: Save the parameters passed (i.e. Entity number and dbkey)

Step 2: Extract the required starting address of the pasrec array from the Index number (i.e. entity number passed.)

Step 3: Move the record addressed by the given dbkey from datastore, into a temporary buffer.

Step 4: Do a block transfer from temporary buffer to pasrec array buffer, less the header and the name words. (2 words if name is a character

string or 1 word otherwise). Stop.

## 6.3 Algorithm for Filling Successive Records

In the case of variant information, there may be more than one entity that satisfies the given conditions. For instance, if we are looking for bridges on a certain road that satisfy certain given conditions like, length, span, material etc., then, we have to access more than just one record. In such cases, we transfer all the bridges on that entity into the pasrec array starting from a known address (depending on Entity number passed as a parameter). Thereafter, by the process of checking and elimination we flash back all those bridges which satisfy the given conditions and reject the rest.

Step 1: Save the parameters passed (dbkey of the first occurrence of that record and the Index number of the Pasrecarray.

Step 2: Transfer the first record into the Temporary buffer. Save contents of the last word in a Register.

Step 3: Do a block transfer into the pasrec array starting from the address extracted (since index number of the Pasrecarray is passed as a parameter), less the header, name and last word.

Step 4: If last word saved is $\neq -1$, go to Step 2 after setting the dbkey of first occurrence to the contents of last word in register, else stop.

## 6.4 Algorithm for Storing a Record

Each time a record is stored, its dbkey has to be put in the right half of system page if it is indeed the first occurrence. For subsequent records, the dbkey from system page has to be transferred to the last word (next pointer) of the current record and the new dbkey returned is put in the right half of system page. Then, do a block transfer from Pasrecbuffer to temporary buffer. Call "ENTRDB" to store this record back in the data store.

Step 1: Save the parameters passed, i.e. dbkey, system current and name.

Step 2: From information available in system page, reconstruct header and name info in the appropriate locations in the temporary buffer.

Step 3: Save the value of dbkey returned by "GETDBKEY" in a register.

Step 4: Move the dbkey from the system page into the last word in temporary buffer. Move the dbkey returned by "GETDBKEY" into the Right half of

system page against the entity whose location is pointed by system current.

Step 5: A call to "SAVEREC", ensures the transfer of information from the pasrecbuffer into temporary buffer and further into the datastore by a call to "ENTRDB." Stop.

## 6.5 Algorithm for Coding and Evaluating an Expression

This algorithm to evaluate an expression, is needed while processing the command "LIST". In command "LIST, "AND" and "OR" are the only two logical operators that are allowed. The expression is evaluated from left to right without assigning any priority to the two allowed logical operators. For instance, if the command was:

LIST A or B and (C and D) or E

The above expression would be evaluated as if the given expression was:

(((A or B) and (C and D)) or E)

although no parenthesis were given by the user. However, if parenthesis were given, then the given order would be strictly followed.

For processing the above command, a recursive function called, "ANSWER" is used. An array [1...50 of]

characters, is declared in procedure "QUERY" and is filled while processing the command "LIST".

Step 1: While character is a '(' do I := I+1
Expary [I] := character

Step 2: While character ≠ a logical operator do store the subexpression. Increment 'I' and remember this value.

Step 3: Evaluate this subexpression. If the condition is satisfied, put a '1' in the expary at the location 'I' previously stored, else put a 'Ø' in that location. Increment 'I' again and
Expary [I] := "A" for "AND" or
:= "O" for "OR".

Step 4: Store the value of I in Expsize. Move down the Exparray from I := 1 to I := Expsize. Each time a ')' is met "ANSWER" calls itself back recursively, incrementing 'I' each time till it meets either a '0' or '1'.

Step 5: At this point, the boolean value of answer is set to either 'true' or 'false', depending on the expression it encounters. Store the value of logical operator as "A" or "O". The logical follow of an operator can be either a '(' or another expression.

Step 6: If this follow is '(' update value of right result and go to Step 5.

Step 7: Right result is assigned the value of the current expression. Increment 'I'.

Step 8: Evaluate "ANSWER" with its current value and value of right result with the relevant operator which was stored in Step 5. If I < Expsize, go to Step 4 else Stop.

All the important algorithms have been explained.

## 6.6 Execution Sequence

The present work is primarily a package that has been developed for answering certain types of queries where little or no mathematical computation is involved. Hence, the execution sequence that follows pertains to the query module.

On execution command, the programs asks for the required mode namely, query, Insert or Update. On transfer to the query mode, the programs ask for "MAPNUMBER", "RANGE" and "ENTITY" on the prompt "(1) ?". With this, the defaults are set. The current of the system and the current of the entity are set. The relevant datastore file is opened and the program awaits further input on the query mode prompt "?". Thereafter, any of the valid commands listed at Appendix II are processed and

appropriate answers are flashed back onto the screen.

On encountering an illegal command, an appropriate error message is flashed accompanied by a whistle. When the user wants to end this dialogue, he does so by typing "EXIT". On this command, the current file is closed and control is handed back to DEC-monitor.

# CHAPTER 7

# SUMMARY AND SUGGESTIONS

## 7.1 Summary

The present work has achieved the following:

(i) Development of a non-graphical, geographic, query processor for sand model simulation.

(ii) Identification of a data-structure for storing voluminous map data, to aid efficient access and retrieval.

(iii) Desgin, development and implementation of an interactive query language (IQL).

(iv) Development of algorithms for loading (Insert) the data.

(v) Provision for updating the information contained in the datastore.

(vi) Development of algorithms for query handling.

The package was developed keeping in view the requirements of the defence forces for a computer-aided non-graphical query processor system, for handling diverse queries. All the objectives that were spelt out

at the outset have been achieved. However, from the experience gained during the course of this development, a few suggestions if implemented, would truely make this package an asset to the defence forces.

## 7.2 Suggestions for Future Work

For geographic map applications, the entities and their attributes are invariably permanent in nature. That is to say, an entity once created does not disappear or vanish, except in the case of either a bridge or an airfield. At the most, some of the attributes pertaining to an entity may change. Hence, the present work does not cater for an extensive "DELETE" operation. However rare or remote this requirement may be, a delete facility is still a necessity. A delete module should be developed and integrated along with this package to make it more versatile.

Secondly, Insert and update operations are being done through an input file, presently. There is a requirement for adding on an interactive Insert and Update facility. This requirement is more pronounced in the case of an update operation. When implemented, it would relieve the user of the burden of having to type in all attribute values, irrespective of the fact whether or not a certain attribute value has changed.

In order to save a certain query dialogue, presently the "OPSER" transaction facility of the host system (DEC-MONITOR) is being used. A file storing facility could be added onto this package. On the command "EXIT", the user could be asked, whether he would like to save a copy of his dialogue. If yes, that file could be stored giving it a ".LOG" extension, else it could just be deleted.

Lastly, a small monitor facility could be developed and integrated along with this package. This would enable the user to store his own directory with all files and maps present in the system. Presently, this facility is transparent to the user, as it is being taken care of in the disc I/O routine (MACRO). On specifying a certain map-number and range, the disc I/O has two types of returns. If found, the corresponding file is opened and the system awaits the next input command. In cases where the file is not found, disc I/O executes an error return wherein, the appropriate error message is flashed.

## 7.3 Concluding Remarks

It has been the endeavour to make the present work as versatile and useful as possible. Wherever possible, this package was dveloped keeping in mind that,

this would form the basis for further improvement and future work.

One overwhelming factor all through the design and development phase has been, to make this package as amenable to change as could be helped. When augmented with the above suggestions, the present work would truely form a solid base and open up new vistas for future work. When fully developed, it would be a rare asset to the fast modernizing defence forces of the nation.

## REFERENCES

1. Major L.K. Chopra, 'Computer Aided Sand Model Picture Creation Using Interactive Mode', M.Tech. thesis submitted to Computer Science Department, IIT, Kanpur, August 1981.

2. Capt A.V. Subramanian, 'Cartographic Considerations in Computer-Aided Sand Model Simulations', M.Tech. thesis submitted to Computer Science Department, IIT, Kanpur, December 1982.

3. Dana, H. Ballard, 'Strip Trees: A Hierarchical Representation for Curves', paper published in communications of the ACM, May 1981, Vol 24, Number 5, pp 310-321.

4. Jeffrey, D. Ullman, 'Principles of Database Systems,' Computer Science Press, Inc, 1980.

5. 'AQL - A Relational Database Management System and Its Geographical Applications' and 'Some Database Requirements for Pictorial Applications', Lecture Notes in Computer Science, Springer-Verlag, Series 81, 1979.

6. Girish, C. Elchuri, Sundar J. Vaska, Kompella V. Rao, 'Design and Implementation of GDB-10', M.Tech thesis submitted to Computer Science Department, IIT, Kanpur July 1983.

7. Capt D.S. Virk, 'Query Processing From Military Maps and Sand Models (Graphical Queries)', M.Tech. thesis submitted to Computer Science Department, IIT, Kanpur, August 1983.

## APPENDIX I

(Refer to Section 2.3, Page 12)

## ENTITIES AND THEIR ATTRIBUTES

| S. NO. | ENTITY | ATTRIBUTES | |
|---|---|---|---|
| | | NON-VARIANT | VARIANT |
| 1. | ROAD | Road Type | Bridge |
| | | All Weather | Level-Crossings |
| | | Metal | Village |
| | | Defence | |
| | | Load Capacity | |
| | | Lanes | |
| 2. | RIVER | Seasonal | Bridges |
| | | Direction | Drinkability info |
| | | Width | Riverbed info<br>Slope info |
| 3. | CANAL | Origin | Bridges |
| | | Destination | |
| | | Lined/Unlined | |
| | | Canal Road | |
| | | Width | |
| | | Bank Slope | |

| | | | |
|---|---|---|---|
| 4. | Railway Line | Guage | Bridges |
| | | Traction | Stations |
| | | Tracks | |
| 5. | Railway Station | Watering | Nil |
| | | Junction | |
| | | Shuntyard | |
| | | Six-hourly Frequency | |
| | | Platforms | |
| | | Ramps | |
| | | Location | |
| 6. | BRIDGE | Load Capacity | Up Entity |
| | | Span | Down Entity |
| | | Width | |
| | | Temporary | |
| | | Defence | |
| | | Overhead Clearance | |
| | | Length | |
| | | Material | |
| | | Location | |
| 7. | VILLAGE | Hostility | |
| | | Airfield | |
| | | Helipad | |
| | | Port | |
| | | Station | |

| | | | |
|---|---|---|---|
| | | Hospital | |
| | | Police Station | |
| | | Microwave Tower | |
| | | Post Office | |
| | | Telegraph Office | |
| | | State Capital | |
| | | District Headquarters | |
| | | Microwave Repeater | |
| | | Power House | |
| | | Waterplant | |
| | | Plant Capacity | |
| | | Telephone Exchange | |
| | | Exchange Capacity | |
| | | Trunk Lines | |
| | | Population | |
| | | Mean Sea Level | |
| | | Location | |
| 8. | WATER RESOURCES | Location | Wells |
| 9. | AIR-FIELD | Defence | |
| | | Temporary | |
| | | Hangar Facility | |
| | | Fuel Facility | |
| | | Underground | |
| | | Repair Facility | |
| | | Comn. To | |

Nearest Town

Obstacles

Runway

Runway Info

Location

## APPENDIX II

(Refer to Section 3.0, Page 15)

## IQL COMMANDS

1. :I - Transition to the Insert mode
2. :U - Transition to the Update mode
3. :D - Transition to the Delete mode
4. :Q - Transition to Query mode
5. :K - Exit from program
6. ENTITY: entityname - for setting entity default and initializing system current.
7. HISTORY - Flashes the complete history of the entity whose index is equal to the value of system current.
8. LIST - Flashes the list of all those entities which satisfy the given expression that follows command "LIST" within the default range set.
9. LISTALL - Flashes the list of all the entities in the default range set. No expression can follows the command "LISTALL".
10. NUMBER - Returns an integer which is the total of all these entities

- which satisfy the given expression that follows command "LISTALL" with the specified default range.

11. Numberall - Returns an integer which is the total of all these entities in the specified range. No expression can follows command "NUMBERALL".

12. AGAIN - Reinterprets the current command.

13. ENTITIES - Lists out all the entities present in package.

14. ATTRIBUTES - Lists out all the attributes of the entity that is typed after command "ATTRIBUTES" preceded by a slash ('/').

15. HELP - Prints out the contents of the help file.

16. EXIT - Transfer of control from query module to main.

## APPENDIX III

(Refer to Section 3.0, Page 15)

## ERROR MESSAGES

1. This entity occurrence is not defined in database.
2. Syntax Error! ':' expected after entity .
3. Syntax Error! '/' expected after verbs LIST, HISTORY AND ATTRIBUTES
4. Illegal character.
5. Non-existent attribute for entity under question.
6. Syntax Error! Relational operator expected.
7. Illegal word.
8. Command incomplete.
9. This command illegal for this entity.
10. This is not a variant attribute. Check!
11. Illegal type against value. Check!
12. Illegal value assignment to a boolean type.
13. Illegal character detected. Only digits are allowed.
14. Expression incomplete. ')' expected.
15. Expression incomplete. '(' parenthesis expected.
16. '"' or '#' expected before giving entity name in insert/update.

17. Cannot insert in database. Entity already present.

18. This entity has no variants.

19. Cannot update. This entity does not exist in database.

20. ERROR in input command. Illegal mode.

## APPENDIX IV

(Refer to Section 4.1.3, Page 28)

## SIZE CALCULATIONS - LINES AND RECORDS

| | | | |
|---|---|---|---|
| Number of words occupied by | Road | = | 7 |
| do | River | = | 8 |
| do | Canal | = | 9 |
| do | Rlyline | = | 5 |
| do | Rlystns | = | 7 |
| do | Bridge | = | 13 |
| do | Village | = | 11 |
| do | Waterres | = | 4 |
| do | Airfd | = | 24 |
| | Total | | 88 |

| | | | |
|---|---|---|---|
| Number of words occupied by | Bridgerec | = | 18 |
| do | Inforec | = | 26 |
| do | Lxingrec | = | 22 |
| do | Miscrec | = | 22 |
| | Total | | 88 |

| | | | |
|---|---|---|---|
| Total number of entities | | = | 9 |
| Total number of arrays | | = | 4 |
| | Total | | 13 |

| | | | |
|---|---|---|---|
| Number of words occupied by | Entities | = | 88 |
| do | Headers and Name | = | 36 |
| do | Arrays | = | 88 |
| | Total | | 212 |

Average word length = 212/13

= 16 (approx)

Page size (2 blocks of DEC-10) = 256

Let number of lines/page = x

$\therefore$ number of page headers = 1

number of line headers = x

$\therefore$ $1 + x + 16x = 256$

$\therefore$ $x = 255/17 = 15$

$\therefore$ number of lines/page = 14

| | | | |
|---|---|---|---|
| Bits required for | line count | = | 4 |
| do | words left | = | 8 |
| do | first free word | = | 8 |
| do | page flag | = | 1 |

Maximum record length (words) = 24

Bits required for record length (reclen) = 5

Note: Header field sizes have been worked out based on the above calculations.

## APPENDIX V

(Refer to Section 5.3.2, Page 48)

## INPUT FILE: DATAIN

```
ROAD "NH1"
0
T
T
F
100.0
4
RIVER "KALINADI"
F
S2N
30.0

CANAL "RAJASTANCA"
MALI
KOT
T
T
55.0
60
RLYLINE "SRINAGAR RL"
0
T
2
RLYSTNS "JAMMU"
T
F
T
2
6
2
72.9  67.2
BRIDGE  1
200
10.5
5.5
F
F
5.5
55.0
IRON
"SRINAGARRL"
"RAJASTANCA"
72.5  66.8
```